Inhalt

1 – Prozessverwaltung: Einplanungsgrundlagen

1.1 – Programmfaden

1.1.1 – Grundsätzliches

Der **Programmfaden** (engl. ~~Fred~~ Thread) ist die grundlegende Zuteilungseinheit eines Prozessors. Er bezeichnet einen einzelnen Fluss aus ausgeführten Befehlen eines Programms. Damit Threads eingelastet werden und auf dem Prozessor laufen können, benötigen sie alle zur Ausführung erforderlichen Betriebsmittel.

Dabei ist zu beachten:

- Betriebsmittel können nur begrenzt zur Verfügung stehen (CPU-Zeit selbst ist ein Betriebsmittel in diesem Sinne).
- Daher können auch nicht einfach so alle Threads gleichzeitig „Zu Ende“ berechnet werden.
- Das Betriebssystem organisiert die zeitliche Planung und Einlastung von verschiedenen Threads in den Prozessor in Abhängigkeit der verfügbaren Betriebsmittel (Scheduling).

1.1.2 – Threadverläufe

CPU-Burst: Ein zeitlich kontinuierlicher Strang an ausgeführten Befehlen eines Threads, während dieser in der CPU aktiv bearbeitet wird. Ein CPU-Burst beginnt, wenn ein Thread eingelastet wird und endet, wenn dieser blockiert oder terminiert.

I/O-Burst: Ein Thread erfordert zur Weiterarbeit Informationen aus CPU-unabhängigen Komponenten des Rechners (Peripherie oder Betriebsmittel die von anderen Threads beansprucht werden). In dieser Zeit kann der Thread nicht weiter ausgeführt werden.

Moderne multi-programmierte Betriebssysteme orientieren ihre Einplanung und Einlastung von Threads an folgendem Kontrollfluss:

1. **Actuate:** Ein laufender Thread erteilt einen I/O-Burst, da er zur Weiterarbeit Betriebsmittel oder Informationen benötigt, die ihm aktuell nicht zur Verfügung stehen.
2. **Schedule:** Der Thread wird blockiert und wartet nun Passiv auf die angeforderten Betriebsmittel.
3. **Dispatch:** Währenddessen wird ein anderer eingeplanter, laufbereiter Thread in den Prozessor eingelastet.
4. **Interrupt:** Sobald das geforderte Betriebsmittel verfügbar ist, wird dies dem Betriebssystem durch das I/O-System[1] oder eben entsprechende andere Prozesse signalisiert.
5. **Schedule:** Der blockierte Thread wird in den Zustand „bereit“ gesetzt und zur weiteren Ausführung eingeplant.
6. **Dispatch:** Sobald der bereite Thread gemäß des Schedulings wieder ausgeführt werden darf, wird er in die CPU eingelastet und arbeitet weiter

[1] Die Peripherie des Computers wird von der CPU aus logischer Sicht als eigener Prozess angesehen.

1.1.3 – Leistungsoptimierung

Indem man die Zeit, in der Threads passiv auf ihre Betriebsmittel warten müssen, für die Ausführung anderer Threads nutzt, kann man die CPU und die Peripherie sehr viel besser zeitlich auslasten.

Sind weniger Prozessoren als Threads vorhanden, so müssen die Threads serialisiert werden.
Folgende Formel beschreibt (Im Bezug eines verfügbaren CPU-Kerns), wie sich die absolute Ausführungsdauer von laufbereiten Threads mit fester Bearbeitungsdauer k proportional zur Threadanzahl vergrößert:

$$\frac{1}{n} \cdot \sum_{i=1}^{n} (i-1) \cdot k = \frac{n-1}{2} \cdot k$$

Dabei ist n die Anzahl der Threads, wobei jeder Thread die Bearbeitungsdauer k besitzt. Jeder i-te Thread wird dabei um die Zeitdauer $(i-1) \cdot k$ verzögert.

Die Threadverzögerung wird dominiert von den Wartezeiten auf die I/O und nicht von den Wartezeiten auf die CPU. Überlast durch zu viele Threads muss vermieden werden.

1.2 – Arbeitsweisen

1.2.1 – Zuteilungsebenen- und Übergänge

Zuteilungsentscheidungen des Betriebssystems für Prozesse können von unterschiedlicher Dauer geprägt sein und lassen sich kategorisieren. Je nachdem, welche Einplanungsebene man betrachtet, können Prozesse im Rechner verschiedene logische Zustände annehmen.

Short-term Scheduling (Kurzfristige Einplanung) [µs - ms]
Der Short-term Scheduler ist zuständig für die Einplanung von Threads zur Einlastung in die CPU gemäß der Verfügbarkeit und logischen Abhängigkeit der relevanten Betriebsmittel. Er entscheidet, welcher Thread als nächstes ausgeführt wird.

Ein Betriebssystem, welches Mehrprozessbetrieb ermöglichen möchte, muss obligatorisch ein Short-term Schedulingverfahren implementieren. Der Mehrprozessbetrieb wird durch die Serialisierung von Threads ermöglicht.

Prozesse können in dieser Ebene folgende Zustände annehmen:

- Bereit (ready): Der Prozess steht auf der *ready list* und ist bereit zur Ausführung durch den Prozessor. Seine Position auf der Liste wird vom Einplanungsalgorithmus bestimmt.
- Laufend (running): Der Prozess wurde in die CPU eingelastet und wird aktuell ausgeführt. Der Fred vollzieht seinen ~~Becken~~ CPU-Stoß. Pro CPU gibt es zu jedem Zeitpunkt nur einen laufenden Prozess.
- Blockiert (blocked): Der Prozess erwartet die Zuteilung von Betriebsmitteln und kann deswegen nicht weiter ausgeführt werden.

Medium-term Scheduling (Mittelfristige Einplanung) [ms - s]
Der Medium-term Scheduler gestaltet und plant das Laden von Prozessen aus dem Hintergrund- in den Hauptspeicher und umgekehrt (swapping). Dadurch wird der Mehrprogrammbetrieb verbessert, da der

Datenkontext von aktuell nicht zur Einlastung geeigneten Prozessen den Hauptspeicher nicht unnötig beanspruchen muss und dieser stattdessen von wichtigeren Prozessen eingenommen werden kann.

Prozesse können in dieser Ebene folgende Zustände annehmen:

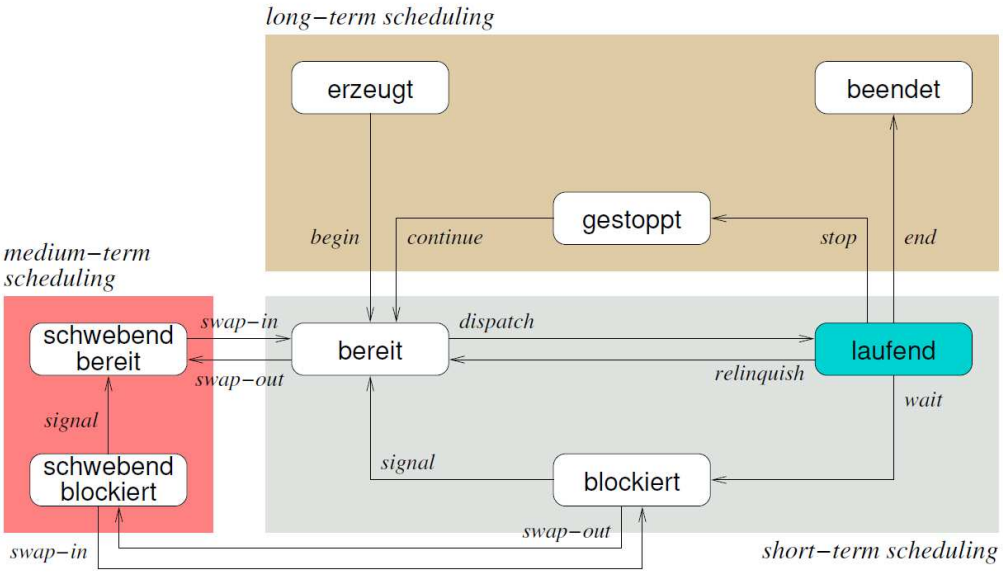
- Schwebend bereit (ready suspend): Die Prozessinkarnation ist ausgelagert und wartet auf die Einlagerung in den Hauptspeicher. Keine Threads des zugehörigen Programms sind im Moment eingelastet.
- Schwebend blockiert (blocked suspend): Der Prozess wird blockiert, da er auf ein Ereignis (Betriebsmittel, I/O) warten muss und wurde deswegen ausgelastet. Er wartet im Hintergrundspeicher auf den Ereigniseintritt und den darauffolgenden Übergang in den „Schwebend bereit"-Zustand.

Long-term Scheduling (Langfristige Einplanung) [s - min]
Auch Job-Scheduling genannt. Der Long-term-Scheduler wählt Programme aus der Warteliste aus und generiert aus ihnen Prozesse (und beendet diese auch). Das Hauptziel des Long-term Schedulers ist es, eine gerechte und ausgeglichene Prozessteilnahme zu gewährleisten, indem der Mehrprogrammbetrieb kontrolliert und ggf. eingeschränkt wird. Prozesse, die aus Programmen generiert wurden, können betriebssystembedingt entweder sofort im Prozessor eingelastet werden oder dem Medium- bzw. Short-term Scheduler zugeteilt werden.

Prozesse können in dieser Ebene folgende Zustände annehmen:

- Erzeugt (created): Von einem Programm wurde ein Prozessexemplar generiert, welches jetzt fertig zur Verarbeitung ist. Es steht gegebenenfalls noch eine Zuteilung des Prozesses an den Hauptspeicher aus.
- Gestoppt (stopped): Der Prozess wurde aus irgendeinem Grund angehalten (Überlast, Verklemmungsprävention, ...). Der Prozess wartet auf erneute Einlastung oder Terminierung.
- Beendet (ended): Der Prozess ist zu Ende und erwartet seine Entsorgung, u.a. Freigabe von allen belegten Betriebsmitteln.

Abfertigungszustände im Zusammenhang visualisiert:



1.2.2 – Verdrängung

Das Betriebssystem kann die Möglichkeit haben, Prozesse dazu zu bewegen, wiederverwendbare Betriebsmittel (typischerweise die CPU) frühzeitig an andere Prozesse abzugeben (Also ohne einen I/O-Burst). Diesen Vorgang nennt man **Verdrängung**.

Wird ein Prozess verdrängt, so bedeutet dies konkret:

1. Es tritt ein besonderes Ereignis (Verdrängungsaufforderung) ein, welches bewirkt, dass der verdrängende Prozess von „blockiert" in den Zustand „bereit" überführt wird.
2. Das Ereignis unterbricht auch den aktuell laufenden (zu verdrängenden) Prozess, welcher infolgedessen aus dem Zustand „laufend" in den Zustand „bereit" überführt wird.
3. Der verdrängende Prozess wird ausgewählt und zur Einlastung in die CPU aufgenommen.
4. Der verdrängte Prozess wird wieder eingeplant um später erneut eingelastet werden zu können.

Das Scheduling vom Betriebssystem plant die Einlastung von Prozessen nicht immer zeitnah mit Verdrängungsaufforderungen ein. Aus diesem Grund kann das Betriebssystem eine Verdrängungssperre für Prozesse implementieren, welche kritische Abschnitte schützt indem eine Verdrängung temporär unmöglich gemacht wird.

Die Verdrängung von Prozessen kann Betriebssystembedingt zu einer zeitlichen Verzögerung des Ausführungszyklus führen. Es entstehen **Latenzzeiten**.

Einplanungslatenz (scheduling latency)
Die Zeit, in der das Betriebssystem die Einplanung oder Umplanung von Prozessen berechnet. Die Dauer der Einplanungslatenz ist unter Umständen vorhersehbar (predictable) und deterministisch und ist somit zeitlich nahezu konstant. Um die Interferenz durch Verdrängung möglichst klein zu halten, sollte die Einplanungslatenz von kurzer Dauer sein.

Einlastungslatenz (dispatching latency)
Die Zeitspanne zwischen Einplanung und Einlastung eines Prozesses. Insbesondere ist damit die Zeit gemeint, in der sich der Prozessor bei einem Kontextwechsel im Untätigkeitszustand befindet. Die Dauer der Einlastungslatenz ist im Allgemeinen vorhersehbar und deterministisch. Man unterscheidet in diesem Bezug zwischen zwei Arten von Verdrängung, bei denen die Einlastungslatenz von unterschiedlicher Dauer ist.
Ereignisbasierte Betriebssysteme lassen die Einlastung von Prozessen nur an bestimmten, fest definierten Zeitpunkten zu, sogenannten *Verdrängungspunkten (preemption point)* → Größere Latenz.
Prozessbasierte Betriebssysteme lassen Einlastung jederzeit zu, sie arbeiten *voll verdrängend (full preemptive)* → Kleinere Latenz.

1.3 – Gütemerkmale

1.3.1 – Grundsätzliches

Für die Auswahl einer geeigneten Schedulingstrategie beim Entwurf eines Betriebssystems können verschiedene Kriterien ausschlaggebend sein. Man unterscheidet zwischen **benutzerorientierten Kriterien** und **systemorientierten Kriterien.**

Diese Kriterien schließen sich nicht aus. Es gilt, abhängig von der Anwendungsdomäne des Betriebssystems, beim Entwurf auf bestimmte Kriterien einen Schwerpunkt zu setzen.

1.3.2 – Benutzerdienlichkeit

Der Fokus liegt auf der Dienlichkeit für den Benutzer. Entsprechend muss die Prozesseinplanung so strukturiert sein, dass das Systemverhalten, welches vom Benutzer wahrgenommen wird, für diesen möglichst zufriedenstellend ist. Benutzerorientierte Kriterien bestimmen im großen Maße, wie das System beim Benutzer „ankommt".
Charakteristische Anforderungsmerkmale sind:

- Antwortzeit: Minimierung der Zeitdauer von der Auslösung eines Systemaufrufs bis zur Entgegennahme der Rückantwort. Gleichzeitig soll die Anzahl der interaktiven Prozesse zu einem gegebenen Zeitpunkt maximiert werden.
- Durchlaufzeit: Minimierung der Zeitdauer vom Starten eines Prozesses bis zu seiner Beendigung. Die effektive Prozesslaufzeit und alle damit anfallenden Wartezeiten sollen möglichst kurz gehalten werden.
- Termineinhaltung: Fristgerechtes Starten und/oder Beenden eines Prozesses bis zu einem fest vorgegebenen Zeitpunkt.
- Vorhersagbarkeit: Deterministische Ausführung eines Prozesses unabhängig von der jeweils vorliegenden Systemlast.

Je nach Anwendungsdomäne können diese Merkmale eine unterschiedliche Wichtung haben.

1.3.3 – Systemperformanz

Hier liegt der Fokus auf der Systemperformanz. Die Prozesseinplanung soll sich an der effektivsten und effizientesten Auslastung aller Betriebsmittel orientieren. Systemorientierte Kriterien bestimmen im weiteren Sinne die Wirtschaftlichkeit und Nützlichkeit des Systems.
Wünschenswerte Anforderungsmerkmale sind:

- Durchsatz: Für eine Vorgegebene Zeiteinheit soll die Anzahl vollendeter Prozesse maximiert werden. Die effektiv im System geleistete Arbeit soll so hoch wie möglich sein.
- Prozessorauslastung: Maximierung des Prozentanteils der Zeit, in der die CPU Maschinenbefehle ausführt. Die Untätigkeitsphasen der CPU während dem Kontextwechsel von Prozessen und sonstigen Wartezeiten soll so kurz gehalten werden wie möglich.
- Gerechtigkeit: Gleichbehandlung der Prozesse. Jeder Prozess der gleichen Prioritätsklasse soll innerhalb eines gewissen Zeitraums seinen nötigen Betriebsmitteln, insbesondere der Einlastungszeit in die CPU, zugeteilt werden.
- Dringlichkeit: Prozesse mit höherer Priorität sollen Vorrang bei der Behandlung erhalten.
- Lastausgleich. Die Betriebsmittel und Komponenten des Rechners sollen möglichst gleich ausgelastet sein. Dies fordert gegebenenfalls die Vorzugbehandlung von Prozessen, die stark belastete Betriebsmittel eher selten und schwach belastete Betriebsmittel eher häufig belegen.

Die Auslegung dieser Merkmale muss konform zur gegebenen Anwendungsdomäne sein.

1.3.4 – Betriebsart

Die Erfüllung von bestimmten benutzerorientierten und/oder systemorientierten Kriterien beim Systementwurf hinterlässt oft einen Rückschluss auf die Rechnerbetriebsart, die man implementieren möchte. Unterschiedliche Rechnerbetriebsarten implizieren die Erfüllung von bestimmten Kriterien und umgekehrt.

Universalbetrieb	⇆	Gerechtigkeit, Lastausgleich
Stapelbetrieb	⇆	Durchsatz, Durchlaufzeit, Prozessorauslastung
Dialogbetrieb	⇆	Antwortzeit
Echtzeitbetrieb	⇆	Dringlichkeit, Termineinhaltung, Vorhersagbarkeit *[!] Steht oft im Konflikt mit Gerechtigkeit/Lastausgleich*

Manchmal ist es auch sinnvoll für bestimmte Prozesse ein künstliches Laufzeitverhalten zu simulieren, welches der inhärenten Vorstellung des Benutzers über die Prozessdauer entspricht, zu Gunsten der Nutzerakzeptanz.

2 – Prozessverwaltung: Einplanungsverfahren

2.1 – Einordnung

Die Anzahl der existenten Schedulingalgorithmen ist beachtlich. Wie bereits untersucht wurde, gibt es für ein Betriebssystem diverse Anwendungsdomänen, gemäß welchen unterschiedliche Kriterien bzw. Anwendungsziele priorisiert werden müssen. Das Betriebssystem muss, im Versuch diesen Zielen nachzukommen, möglichst geeignete Einplanungsverfahren für Prozesse anwenden.

Zur Untersuchung von verschiedenen Schedulingverfahren/algorithmen, ist es zunächst sinnvoll, sich ein Schema zu überlegen, nach welchem man seine Untersuchungen charakterisieren kann.

2.1.1 – Souveränität

Das Betriebssystem kann sich bei der Einplanung von Prozessen unterschiedlich nachlässig gegenüber der Betriebsmittelverteilung für verschiedene Prozesse verhalten, insbesondere der CPU-Einlastungszeit. Je nach bevorzugten Gütemerkmalen kann es dabei von Interesse sein, die Häufigkeit der Programmunterbrechungen und der damit verbundenen Umlastungen von bestimmten Prozessen zu vermeiden oder zu begünstigen.
In diesem Bezug unterscheidet man **kooperative** und **präemptive Planung**.

Kooperative Planung (cooperative scheduling)
Ein Prozess kann die CPU nur dann für einen anderen Prozess abgeben, wenn dieser logisch von ihm abhängig ist. Der Prozess muss dafür in irgendeiner Form mit dem anderen Prozess kooperieren und auf Betriebsmittel angewiesen sein, die ihm nur der andere Prozess zur Verfügung stellen kann. Ist der Prozess unabhängig, dann kann das Betriebssystem ihn nicht von alleine unterbrechen und er wird so lange ausgeführt, bis er abgeschlossen wurde oder ein besonderer Systemaufruf getätigt wird. Die Systemaufrufbehandlung aktiviert dann den Scheduler.
Kooperative Planung ermöglicht und begünstigt *CPU-Monopolisierung*: Ein Prozess oder eine Menge von abhängigen Prozessen kann bis zur Vollendung (mit Uneingeschränkter CPU Auslastung) komplett „an einem Stück“ ausgeführt werden. Das birgt jedoch auch die Gefahr, dass eine systemaufruffreie Endlosschleife die CPU ewig beansprucht und somit keine anderen Prozesse je zur Ausführung kommen können.

Präemptive Planung (preemptive Scheduling)
Das Betriebssystem ist bemächtigt, den aktuell eingelasteten Prozess jederzeit aus der CPU zu verdrängen. Dabei ist es völlig egal, ob dieser von anderen Prozessen abhängig ist oder nicht.
Die Entscheidung zur Verdrängung wird von besonderen Ereignissen angesteuert, wie etwa die Ankunft eines Prozesses mit höherer Priorität oder eine asynchrone Programmunterbrechung. Die Behandlung dieses Ereignisses aktiviert dann den Scheduler.
Präemptive Planung verhindert *CPU-Monopolisierung*. Endlosschleifen verhindern die Ausführung von anderen Prozessen nicht, da der „hängende“ Prozess jederzeit verdrängt werden kann.

2.1.2 – Transparenz

Wenn das Betriebssystem ein bestimmtes Einplanungsverfahren verwendet, dann kann man, je nach algorithmischer Beschaffenheit des Verfahrens, den Grad der Gewissheit über den zukünftigen Ablauf des Verfahrens bewerten. Ein Einplanungsverfahren kann sich bei seiner Planung unterschiedlicher Mechanismen bedienen. Unter diesen Mechanismen gibt es solche, die in ihrer Funktionalität transparent und somit vollkommen vorhersehbar sind, und auch solche, die von zufallsbestimmten Variablen abhängen und dadurch unberechenbar handeln.
Man unterscheidet in diesem Sinne zwischen **deterministischer** und **probabilistischer Planung**.

Deterministische Planung (deterministic scheduling)
Ein deterministisches Planungsverfahren funktioniert so, dass alle Prozesse, Prozesseigenschaften (CPU-Stoßlänge) und synchronen Ereignisse im Voraus bekannt sind. Die Vorgehensweise des Verfahrens ist eindeutig festgelegt und kann vorhergesehen werden. Eine genaue Prognose der CPU-Auslastung sowie der Einplanungs- und Einlastungslatenzen ist möglich.
Das Betriebssystem ist dadurch auch in der Lage, festgelegte Zeitgarantien einzuhalten.

Probabilistische Planung (probabilistic scheduling)
Ein probabilistisches Planungsverfahren bedient sich bei der Einplanung von Prozessen wahrscheinlichkeitsbedingten Mechanismen. Die Prozesse, Prozesseigenschaften und Termine sind nicht vorhersehbar. Prognosen über CPU-Auslastung, Latenzen und CPU-Stoßlängen sind nicht möglich, lediglich grob abschätzbar.
Das Betriebssystem ist nicht in der Lage, festgelegte Zeitgarantien einzuhalten. Die Anwendung trägt darüber Verantwortung.

Hier gibt es nicht zwingend eine eindeutige Abgrenzung. Einplanungsverfahren können sich sowohl deterministischer als auch probabilistischer Elemente bedienen.

2.1.3 – Dynamik

Ein Einplanungsverfahren für Prozesse kann einen unterschiedlichen Grad der Kopplung mit dem ausgeführten Programm, das mit diesen Prozessen verknüpft ist, aufweisen. Das Scheduling kann verschiedenartig stark von der Ausführung der planmäßig eingeteilten Prozesse abhängen.
Hier unterscheidet man zwischen **vorlaufender** und **mitlaufender Planung**.

Vorlaufende Planung (off-line scheduling)
Die Einplanung von Prozessen geschieht vollständig vor der Inbetriebnahme der Prozesse bzw. des Rechensystems. Das Verfahren verwendet alle vor der Ausführung des Prozesses verfügbaren Informationen, um daraus einen vollständigen Ablaufplan zu generieren, beispielsweise durch Quelltextanalyse des Programms.
Vorlaufende Einplanungsverfahren sind häufig begrenzt auf strikte Echtzeitsysteme.

Mitlaufende Planung (on-line scheduling)
Die Einplanung von Prozessen geschieht während der Ausführung der Prozesse. Zumeist werden solche Verfahren von Stapelsystemen, interaktiven Systemen, verteilten Systemen und schwachen und festen Echtzeitsystemen benutzt.

Auch hier gibt es nicht zwingend eine eindeutige Abgrenzung.

2.1.4 – Symmetrie

Sofern es mehrere Prozessorkerne auf dem Rechner gibt, kann das Einteilungsverfahren auch unterschiedlich stark von den wechselseitigen Eigenschaften der einzelnen Kerne bedingt sein.

Ungleichmäßige Planung (asymmetric scheduling)
Der Scheduler verteilt die Prozesse ungleichmäßig auf die vorhandenen Prozessoren gemäß ihrer funktionalen Spezifikation. Das Betriebssystem hat Entscheidungsmacht über die individuelle Prozessorvergabe und bedient sich jeweils einer lokalen Buchführung über die bereiten, lauffähigen Prozesse. Ungleichmäßige Planung ist obligatorisch in einem asymmetrischen Multiprozessorsystem, da das Betriebssystem dort auf einem bestimmten Prozessor ausgeführt wird, wobei dann die restlichen Prozesse systematisch auf die restlichen Kerne verteilt werden müssen.
In einem symmetrischen Multiprozessorsystem wird das Betriebssystem von allen Kernen gleichermaßen ausgeführt. Ungleichmäßige Planung ist bei solchen Architekturen nicht verpflichtend.

Gleichmäßige Planung (symmetric scheduling)
Die Prozesse werden möglichst gleichmäßig auf alle vorhandenen Prozessoren verteilt. Das Betriebssystem verwaltet eine globale Liste der bereiten lauffähigen Prozesse. Parallel laufende Prozesse müssen vom Betriebssystem gegebenenfalls synchronisiert werden.

2.2 – Verfahrensweisen

Im Folgenden werden grundlegende Ansätze von Schedulingverfahren für **Einkernsysteme** betrachtet.

2.2.1 – First Come First Serve (FCFS)

Bei weitem der einfachste Schedulingalgorithmus. Prozesse werden gemäß ihrer Ankunftszeit eingeplant, d.h. der Prozess, der die CPU als erstes anfordert, wird auch als erstes eingelastet.

Das wesentliche Problem dieses Verfahrens gestaltet sich in der mittleren Wartezeit für die Ausführung aller ankommenden Prozesse. Prozesse werden vollkommen ohne Rücksicht auf die Dauer ihrer CPU-Bursts eingelastet, was zur Folge hat, dass besonders lange Prozesse nach ihrer Einlastung die Ausführung von kürzeren Prozessen zeitlich stark nach hinten verschieben. Dies ist bekannt als der **Konvoieffekt**.

2.2.2 – Round Robin (RR)

Funktioniert wie FCFS, hat aber zusätzlich einen Verdrängungsmechanismus.

Jeder ankommende Prozess erhält eine Zeitscheibe (time slice), nach deren Ablauf er automatisch aus der CPU verdrängt wird. Dadurch ergibt sich ein zyklisches time-sharing Verfahren.

Leider ergibt sich auch bei diesem Verfahren ein **Konvoieffekt**, da Prozesse oft „typisch“ je nach Beschaffenheit und Programminhalt ihre Zeitscheibe entweder sehr selten oder nahezu immer vollständig ausnutzen. Insbesondere ist dieser Kontrast deutlich bei CPU-intensiven und I/O-intensiven Prozessen zu beobachten. Es ergibt sich wieder eine Diskrepanz zwischen den Einlastungszeitpunkten.

2.2.3 – Virtual Round Robin (VRR)

Funktioniert wie RR, implementiert jedoch eine Vorzugswarteschlange und variable Zeitscheiben.

- Alle Prozesse, die ihren CPU-Burst vor dem Ablauf ihrer Zeitscheibe beenden, werden in die Vorzugswarteschlange gebracht, wo sie auf eine erneute Einlastung warten.
- Es werden nur so lange Prozesse aus der Bereitliste eingelastet, wie die Vorzugswarteschlange leer ist.
- Wenn es Prozesse sowohl in der Bereitliste als auch in der Vorzugswarteschlange gibt, dann werden die Prozesse aus der Vorzugswarteschlange priorisiert und entsprechend vor denen aus der Bereitliste eingelastet.
- Durch Betriebsmittelmangel blockierte Prozesse erhalten einen Vorteil gegenüber den Prozessen, die so lange brauchen, dass sie aus der CPU zugunsten anderer Prozesse verdrängt werden mussten.

2.2.4 – Shortest Process Next (SPN)

Dieses Verfahren schätzt oder berechnet die **erwartete Bedienzeit** aller Prozesse *à priori* und sortiert die Bereitliste aufsteigend nach diesen Werten. Dadurch werden die Prozesse mit der kürzesten Laufzeit zuerst bedient, wodurch man die allgemeine Antwortzeiten der Prozesse verkürzt.
Zur Abschätzung der CPU-Stoßlänge eines Prozesses wird ein **heuristisches Verfahren** benutzt. Dabei errechnet SPN die erwartete Bedienzeit anhand des Mittelwerts aller n bisher gemessenen CPU-Stoßlängen, wobei die Stoßlängen abhängig von ihrem Alter noch zusätzlich gewichtet werden müssen. Daraus ergibt sich folgende Formel für die CPU-Stoßlänge:

$$S_{n+1} = \alpha \cdot T_n + (1 - \alpha) \cdot S_n$$

Wobei T_n die zuletzt gemessene CPU-Stoßlänge, S_n die zuletzt geschätzte CPU-Stoßlänge und α mit $0 < \alpha < 1$ ein konstanter Wichtungsfaktor ist.

SPN hat die Nachteile, dass die Prozesse mit sehr langer Laufzeit durch konstante Betriebsmittelverweigerung verhungern können *(starvation)* und die Abschätzung der Laufzeit trotz der verfügbaren Heuristik oft ungenau ist.

2.2.5 – Highest Response Ratio Next (HRRN)

Ähnlich wie SPN.
Prozessen wird gemäß ihrer erwarteten Bedienzeit eine Priorität zugeordnet, nach welcher sie in der Bereitliste sortiert werden. Zusätzlich dazu wird periodisch (angesteuert durch einen Uhrtick) ihre Wartezeit berechnet, welche, je nach Wert, einem Prozess ggf. eine höhere Priorität zuordnen kann. Die Priorität R eines Prozesses und somit seine Position in der Bereitliste berechnet sich nach folgender Formel:

$$R = \frac{w + s}{s}$$

Wobei w die aktuell abgelaufene Wartezeit eines Prozesses und s die (nach SPN-Formel) abgeschätzte Bedienzeit eines Prozesses ist.

2.2.6 – Shortest Remaining Time First (SRTF)

Eine Art Hybrid aus SPN und VRR.
Prozesse werden wieder gemäß ihrer **erwarteten Bedienzeit** eingeplant.

Jedoch werden die Prozesse in unregelmäßigen Zeitabständen **spontan** umgeplant. Nämlich genau dann, wenn die erwartete CPU-Stoßlänge des eintreffenden Prozesses kürzer ist, als die noch verbleibende CPU-Stoßlänge des aktuell eingelasteten Prozesses. Der laufende Prozess wird in diesem Fall verdrängt.
Diese spontane Umplanung ist Ereignisgesteuert und (ggf. voll) verdrängend im Ankunftszeitpunkt eines Prozesses.

Nach der Verdrängung kommt der betroffene Prozess entsprechend der Restdauer seiner erwarteten CPU-Stoßlänge auf die Bereitliste.

2.2.7 – Multilevel Queue (MLQ)

Dieses Einplanungsverfahren unterstützt Mischbetrieb und bedient sich (wie der Name schon sagt) der Datenstruktur einer Multilevel-Queue.

- Prozesse werden nach ihren Eigenschaften, wie etwa Prozesstyp, CPU-Stoßlänge, I/O-Intensität, Adressraumgröße, etc. kategorisiert.
- Die Bereitliste ist eine Multilevel-Queue, aufgeteilt in n „getypte“ Listen, wobei n die Anzahl der verschiedenen Prozesskategorien ist.
- Die Prozesse in all diesen Listen werden jeweils pro Liste mit einem lokalen Verfahren (z.B. SPN, RR, FCFS) separat eingeplant.
- Zwischen den Listen wird eine globale Einplanunggsstrategie definiert. Dabei kann man die einzelnen Listen verschiedenen Prioritätsgraden zuordnen oder die Listen werden durch Zeitmultiplexverfahren abgewechselt.
- Die Zuordnung von Prozessen einer Kategorie geschieht statisch, also vor Betriebsanfang.

2.2.8 – Multilevel Feedback Queue (MLFQ)

Hier werden kurze/interaktive Prozesse stark begünstigt, indem die langdauernden durch das Zeitscheibenverfahren herausselektiert werden.

- Es existiert eine Hierarchie aus mehreren Bereitlisten, die unterschiedliche Prioritätengrade besitzen.
- Erstmalig eintreffende Prozesse werden der Liste mit der höchsten Priorität zugeordnet und nach ihrer Ankunftszeit eingeplant.
- Jeder Prozess erhält wieder eine Zeitscheibe. Überschreitet der CPU-Burst eines Prozesses den zeitlichen Rahmen seiner Scheibe, dann wird der Prozess verdrängt und in eine Warteliste mit geringerer Priorität gesetzt.

- Die Prioritätsebenen arbeiten mit unterschiedlichen lokalen Einplanungsverfahren. Die unterste Ebene handelt gemäß RR, alle anderen gemäß FCFS.

2.2.9 – Bilanz

Die folgende Tabelle stellt die Schedulingverfahren gegenüber und charakterisiert sie nach den vorgestellten Eigenschaften:

	FCFS	RR	VRR	SPN	HRRN	SRTF	FB
Kooperativ	✓						
Verdrängend		✓	✓			✓	✓
Probabilistisch				✓	✓	✓	
Deterministisch	Nur im Echtzeitbetrieb						

MLQ erlaubt die Kombination all dieser Verfahren, jedoch abgestuft und nicht alle zusammen auf derselben Ebene. Dadurch wird letztlich eine **Priorisierung** der Strategien vorgenommen. Auf diese Weise kann man gezielt bestimmten Anwendungsbezogenen Bedürfnissen entgegenkommen.

3 – Prozesssynchronisation: Nichtsequentialität

3.1 – Kausalitätsprinzip

3.1.1 – Kausalordnung

Der Begriff **Nebenläufigkeit** (concurrency) bezeichnet das Verhältnis von nicht kausal abhängigen Ereignissen, also solchen Ereignissen, die sich nicht gegenseitig beeinflussen. Zwei Vorgänge oder Prozesse heißen **nebenläufig**, wenn es möglich ist, diese voneinander unabhängig und somit auch gleichzeitig zu bearbeiten.

- Aus logischer Perspektive sind zwei Aktionen (möglicherweise) dann nebenläufig, wenn die eine Aktion zur Ausführung keine Informationen der anderen Aktion benötigt.
- Aus physischer Perspektive benötigen alle Aktionen einen autonomen Aktivitätsträger (Prozessinkarnation).

Der Begriff **Kausalität** (lat. causa: Ursache) bezeichnet die Beziehung zwischen Ursache und Wirkung, betrifft also die Abfolge aufeinander bezogener Ereignisse und Zustände. Eine Aktion A heißt **kausal abhängig** von einer anderen Aktion B, wenn A Ursache oder Auswirkung von B ist. Aktionen sind in diesem Sinne genau dann nebenläufig, wenn sie nicht kausal abhängig sind.

Man kann Nebenläufigkeit auch als relativistischer Perspektive charakterisieren, indem man sagt, dass ein Ereignis nebenläufig zu einem anderen ist, wenn es keine Rolle spielt, in welchem zeitlichen Zusammenhang beide Ereignisse stattfinden. Das andere Ereignis ist liegt im „*Anderswo*“, also weder in der Zukunft noch in der Vergangenheit. In dieser Charakterisierung ist es gleicherweise auch nicht möglich, dass eines der Ereignisse vom anderen kausal abhängig ist. Es kann jedoch sehr wohl kausal abhängig von dritten Ereignissen sein.

Ein „Im Anderswo anderer Ereignisse liegendes“ Ereignis steht für eine nebenläufige Aktion, sofern:

- Es kein Resultat der anderen Ereignisse benötigt.
- Auf Datenabhängigkeiten gleichzeitiger Prozesse Rücksicht genommen wird.
- Kein Ereignis die Zeitbedingungen der anderen verletzt (Zwingendes Merkmal im Echtzeitbetrieb).

Je nach Art der Beziehung zwischen den Ereignissen/Aktionen in gleichzeitig verlaufenden Prozessen ergeben sich für das Betriebssystem unterschiedliche Herausforderungen:

"ist Ursache von" / "ist Wirkung von" ⇝ **Koordinierung** (implizit/explizit)

"ist nebenläufig zu" ⇝ **Parallelität** (implizit)

3.1.2 – Aktionsfolgen

Mehrere (ggf. nichtsequentielle) Prozesse, die sich durch mehr als eine Aktionsfolge in Raum und Zeit überlappen, heißen **gleichzeitig** (simultaneous).
Damit Prozesse gleichzeitig laufen können, muss das Betriebssystem zur **Simultanverarbeitung** (mutliprocessing) von Programmen fähig sein.

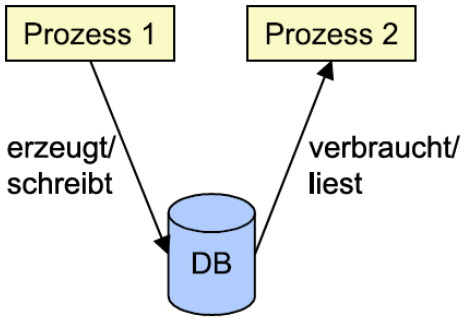
Vertikale Simultanverarbeitung: Das Betriebssystem verarbeitet mehrere Programme „quasiparallel“, indem die zugehörigen Prozesse in den Prozessor durch ein Zeitteilverfahren eingelastet werden.

Horizontale Simultanverarbeitung: Dem Betriebssystem stehen mehrere Prozessoren zur Verfügung. Die Prozesse werden, bedingt durch die Symmetrie des Multiprozessorbetriebs, echtparallel auf mehrere Prozessoren aufgeteilt.

Eine hinreichende Bedingung für die Simultanverarbeitung ist die Verfügbarkeit von Programmen, die mehrere Threads betreiben, darunter fallen Nichtsequentialität in einem Prozess (Parallele Programmierung) oder mehrere sequenzielle Prozesse für ein Programm.

Mehrere gleichzeitige Prozesse, die durch direkte oder indirekte Nutzung gemeinsamer Ressourcen (Betriebsmittel, Variablen) miteinander interagieren, heißen **interaktiv**. Prozesse interagieren miteinander schon allein durch den *Zugriff* auf gemeinsame Ressourcen. Es entsteht eine zeitliche Interferenz, verursacht durch Aktionen, die auf logischer Ebene zeitgleich stattfinden, auf physischer Ebene jedoch sequentiell durchgeführt werden müssen.

Interaktive Prozesse sind **datenabhängig** (Producer/Consumer, Shannon-Weaver Modell,…) und benötigen deshalb Instrumente zur Kommunikation oder Interaktion miteinander, damit ihr Rollenspiel koordiniert werden kann.



3.2 – Sequentialisierung

3.2.1 – Koordinierung

Zwei Aktionen, die kausal abhängig sind, müssen in einer zeitlich festen Abfolge stattfinden.

Die **statische Einplanung** (off-line) von kausal abhängigen Aktionen bestimmt ihre Ablaufreihenfolge vor Inbetriebnahme der beteiligten Prozesse, also vor der Laufzeit. Das Betriebssystem erfordert dafür *à priori* Wissen für eine ausreichende Analyse und **implizite Synchronisierung** der Prozesse.

Die **dynamische Einplanung** (on-line) bestimmt die Ablaufreihenfolge während der Laufzeit kommt ohne Vorabwissen aus. Das Programm ist konstruktiv mit Vorbeugenden Mechanismen ausgestattet, die den Prozess **explizit synchronisieren**.

Beide Aspekte sind vor dem Hintergrund folgender Punkte zu verstehen:

- Der Moment der logisch gleichzeitigen Aktionen ist im Allgemeinen unvorhersehbar.
- Die fragliche Aktion ist komplex, d.h. sie umfasst mehrere Einzelschritte
- Ihre besondere Eigenschaft ist die **Teilbarkeit in zeitlicher Hinsicht**.

Explizite ProzessSynchronisierung verursacht Wettlaufsituationen, insbesondere bei Mitbenutzung derselben wiederverwendbaren Betriebsmittel oder bei Übergabe eines konsumierbaren Betriebsmittels. Der Mechanismus zur Synchronisierung soll *minimal invasiv* auf die Prozesse wirken, also so wenig an ihrem Ablauf verändern wie möglich.

3.2.2 Atomare Aktionen

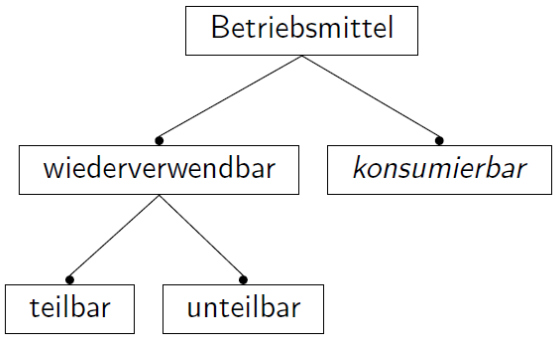
Eine **atomare Aktion** ist eine primitive oder komplexe (aus mehreren Teilschritten aufgebaute) Aktion, die als ein zeitlich untrennbares Ereignis betrachtet werden kann. Ihre Einzelschritte sind logisch unsichtbar und finden im Verbund (scheinbar) gleichzeitig statt.

Die Gleichzeitigkeit (Simultanität) der Einzelschritte wird durch Synchronisierung erreicht. Damit ist gemeint, dass die interaktiven Ereignisse zwischen Prozessen koordiniert werden müssen. Der gleichzeitige Ablauf oder der sequenzielle Ablauf von Aktionsfolgen wird nach einem optimalen Schema explizit festgeregelt.

3.2.3 – Konkurrenz

Betriebsmittel können von Natur aus so beschaffen sein, dass Prozesse in ihrer gleichzeitigen Benutzung eingeschränkt sind. Beispiele für solche Betriebsmittel sind:

- CPU
- Arbeitsspeicher
- Peripheriegeräte
- Signale
- Dateien
- Puffer
- Pageframes



Wenn gleichzeitige Prozesse implizit gekoppelt sind und damit interaktiv sind, dann kann es zwischen ihnen zu einem **Konflikt** kommen, wenn:

- Sie auf ein oder mehrere gemeinsame Betriebsmittel zugreifen möchten
- Diese Betriebsmittel nur begrenzt vorrätig sind
- Diese Betriebsmittel unteilbar sind

Es entsteht eine **Konkurrenzsituation** (contention), wenn einer dieser Prozesse ein Betriebsmittel anfordert, das ein anderer Prozess gerade verwendet. Der Anfordernde Prozess blockiert und muss auf die Freigabe des Betriebsmittels warten. Der Prozess, der das Betriebsmittel beansprucht, muss den passiv wartenden Prozess wieder „deblockieren“.

Ein Betriebsmittel heißt **unteilbar**, wenn es zu einem gegebenen Zeitpunkt nur von einem genau einem Prozess oder Prozessor in Anspruch genommen werden darf. Im Zusammenhang des Betriebssystems bezeichnet der Begriff **Unteilbarkeit** eine Situation, die verhindert, dass Betriebsmittel auf mehrere Prozesse/Prozessoren verteilt werden können.

Aktionen, die auf unteilbare wiederverwendbare Betriebsmittel zugreifen, unterliegen dem wechselseitigen Ausschluss, insbesondere bei der Zuteilung von CPU-Zeit. Die zugreifenden Prozesse müssen **mehrseitig/multilateral** synchronisiert werden und dürfen nicht zeitlich geteilt werden. Der Zugriff muss als atomare Aktion ausgeführt werden.

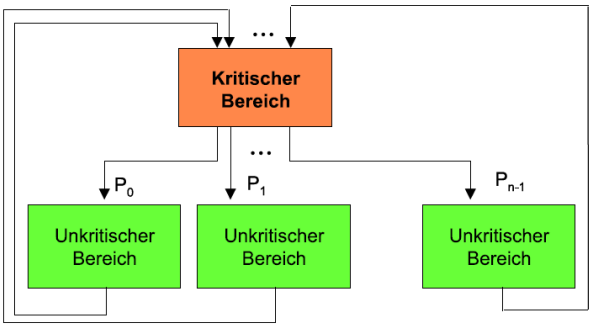
Die Entgegennahme eines konsumierbaren Betriebsmittels wirken nur auf einen Prozess verzögernd. Die gekoppelten Prozesse müssen nur **einseitig/unilateral** synchronisiert werden.

3.2.4 – Wechselseitiger Ausschluss

Ein zentrales Konzept zur Synchronisierung von Aktionen ist der **wechselseitige Ausschluss** (mutual exclusion). Es handelt sich dabei um ein Protokoll, das die Zugriffsaktionen auf ein gemeinsames unteilbares Betriebsmittel sequentialiseren soll.

- Das Betriebssystem sperrt dabei **vor** dem kritischen Ereignis das betroffene Betriebsmittel. Dadurch werden alle Prozesse, die das Betriebsmittel anfordern, in die Blockiertliste gesetzt und erwarten dort die erneute Freigabe des gesperrten Betriebsmittels

- **Nach** dem Abschluss des kritischen Ereignisses wird das gesperrte Betriebsmittel wieder freigegeben. Die blockierten Prozesse durchlaufen erneut das Vergabeprotokoll, was bedeutet dass bei einem weiteren kritischen Ereignis einer der Prozesse das Betriebsmittel in Anspruch nimmt und die anderen Prozesse wieder warten müssen.



Ein zentrales Problem bei der Synchronisierung ist die Intransparenz der tiefer liegenden Abstraktionsebenen beim Entwurf von nebenläufigen Programmen. Es kann nämlich durchaus sein, dass der Programmablauf auf einer Ebene sequentiell und auf einer anderen parallel abläuft. Aus diesem Grund müssen die Instrumente zur Schaffung von kritischen Abschnitten auf Befehlssatzebene (als atomare Operationen für vergleichsweise komplexe Prozeduren) oder auf Maschinenprogrammebene definiert sein.

Zur Modellierung von kritischen Abschnitten eignen sich **Petri-Netze** besonders gut.

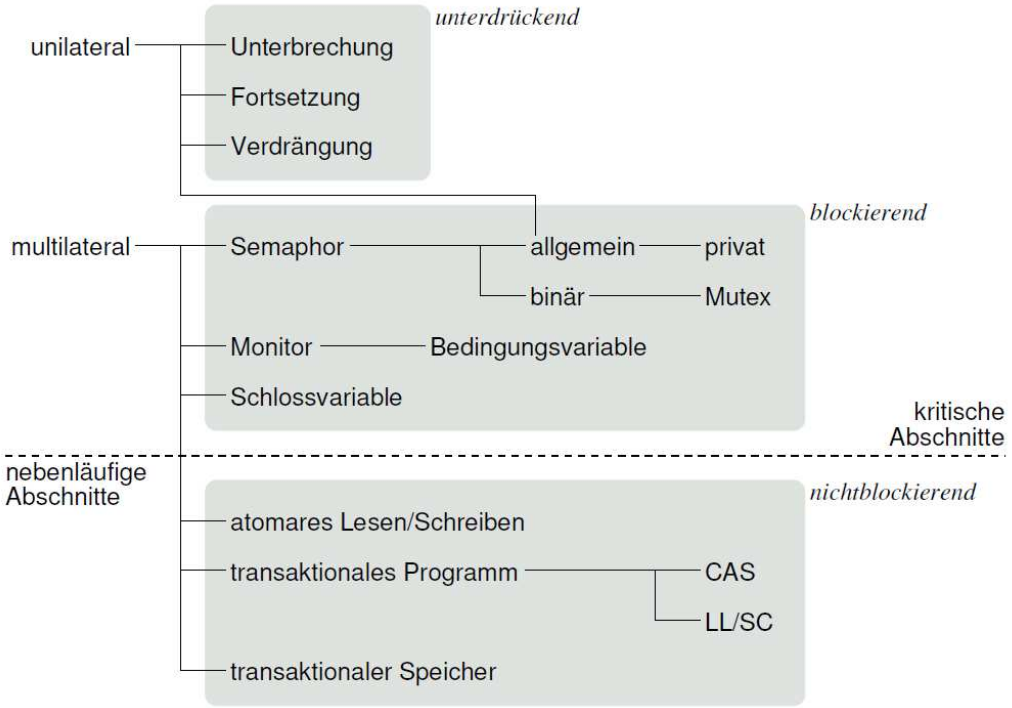
3.3 – Verfahrensweisen

3.3.1 – Einordnung

Die Auswirkungen von Synchronisierungsmechanismen auf die beteiligten Prozesse können je nach Art, Technik und Programmierparadigma sehr verschieden ausfallen. Man kategorisiert die Synchronisationsarten nach folgendem Schema:

- **Unilaterale Synchronisation**: Solche Verfahren wirken sich lediglich auf einen von allen an einem Rollenverhältnis beteiligten Prozessen aus. Die anderen Prozesse schreiten ungehindert fort, ungeachtet des Overheads. Der aktuell eingelastete Prozess wird in der Konkurrenzsituation nicht verzögert Meist werden solche Verfahren benutzt, wenn der Fortschritt eines Prozesses abhängig von einer Bedingung ist, die in einem nichtsequentiellen Programm formuliert ist. (Auch bekannt als Bedingungssynchronisation oder logische Synchronisation).

Man sollte beachten, dass die anderen Prozesse jedoch am Verfahren nicht unbeteiligt sind, da sie immernoch die Verpflichtung haben, die Wartebedingungen für den blockierten Prozess früher oder später aufzuheben.



- **Multilaterale Synchronisation**: Kann sich in einem Rollenverhältnis auf alle beteiligten Prozesse auswirken. Alle betroffenen Prozesse werden vom selben Protokoll bearbeitet, um Synchronisation zu gewährleisten. Bei mehrseitiger Synchronisation ist unbestimmt, welcher der Prozesse ungehindert fortschreitet. Man unterscheidet zwei Ausprägungen:

  - **Blockierende Synchronisation**: Bezeichnet eine Synchronisationsart, in der bestimmte Programmabschnitte gesperrt werden müssen, um einen Konflikt zwischen Prozessen aufzulösen. Es werden Locking-Techniken wie Semaphore oder Mutexe eingesetzt, um kritische Abschnitte zu definieren. Im Regelfall gibt es hier eine zeitlich begrenzte, exklusive Betriebsmittelvergabe. Wird auch als *pessimistische Nebenläufigkeitssteuerung* bezeichnet.

  - **Nichtblockierende Synchronisation**: Gegenteil der blockierenden Synchronisation. Man spart sich den Aufwand der blockierenden Synchronisation dadurch, dass Prozesse ihre Datenstrukturen ausschließlich durch atomare Operationen modifizieren. Solche Synchronisationsverfahren sind eher ungeeignet für die Auflösung von Konflikten im Zusammenhang mit konsumierbaren Betriebsmitteln.

3.3.2 – Fortschrittsgarantien

Der Begriff der **Lebendigkeit** (liveliness) bezeichnet die Eigenschaft eines Systems, trotz sequentialisierung seiner nebenläufigen Aktionsträger niemals zu verklemmen. Dabei ist **Verklemmung** ein Zustand, in dem ein System keinen Fortschritt mehr machen kann, da interaktive Aktionsträger zur

weiteren Ausführung jeweils Betriebsmittel benötigen, welche von anderen verklemmten Aktionsträgern erst bei ihrem Fortschritt freigegeben werden würden.
Über die Lebendigkeit nichtsequentieller Programme kann man im Bezug auf verschiedene Synchronisationsverfahren unterschiedliche Aussagen treffen:

- **Behinderungsfrei** (obstruction-free): Ein einzelner, in Isolation ausgeführter Prozess wird seine Aktionsfolge in einer begrenzen Anzahl an Schritten beenden. Der Prozess findet isoliert statt, sofern alle anderen Prozesse, die ihn behindern könnten, zurückgestellt sind.

- **Sperrfrei** (lock-free): Umfasst Behinderungsfreiheit. Jeder Schritt eines Prozesses trägt dazu bei, dass die Ausführung des nichtsequentiellen Programms insgesamt voranschreitet. Systemweiter Forschritt ist garantiert, jedoch können einzelne Prozesse der Aushungerung unterliegen.

- **Wartefrei** (wait-free): Umfasst Sperrfreiheit. Die Anzahl der zur Beendigung einer Aktion auszuführenden Schritte ist konstant oder zumindest nach oben begrenzt. Systemweiter Fortschritt ist garantiert und Gefahr von Aushungerung ist aufgehoben.

4 – Prozesssynchronisation: Monitore

4.1 – Monitoreigenschaften

4.1.1 – Funktionalität

Ein **Monitor** ist ein Datenstruktur mit **impliziten Synchronisationseigenschaften**. Im konkreten Betrachtungsfall von objektorientierten Programmiersprachen ist ein Monitor eine **thread-sichere** Klasse, was bedeutet, dass einzelne Threads gemeinsam mit den Objekten dieser Klasse arbeiten können, ohne dass es zu einer Verklemmung oder einem ungültigen Programmzustand kommt.
Monitore können auf zwei Arten synchronisiert sein:

Mehrseitige Synchronisation an der Monitorschnittstelle.
Die Schnittstellen der Prozeduren (Methoden) des Monitors sind alle sinnvoll mit kritischen Abschnitten ausgestattet. Ein Prodezuraufruf sperrt den Monitor, beim Abschluss der Prodezur wird der Monitor wieder freigegeben. Die Funktionsweise des kritischen Abschnitts wird vom Kompilierer sichergestellt. Realisiert wird dies zumeist mittels Schlossvariablen oder vorzugsweise Semaphoren.

Einseitige Synchronisation innerhalb des Monitors.
Gekoppelte Prozesse, die auf der Datenstruktur arbeiten, werden bei Bedarf logisch synchronisiert. Die Synchronisationsanweisungen müssen vom Monitor selbst und nicht vom gesamten nichtsequentiellen Programm berücksichtigt werden.
Oft realisiert durch eine Bedingungsvariable und ihre Operationen:

- wait(): Der Prozess wird blockiert und in eine Warteschlange eingefügt.
- signal(): Die Prozesse, die auf diese Bedingungsvariable blockiert sind, wieder freigegeben.

4.1.2 – Klassenkonzept

Wie bereits erwähnt, ist ein Monitor einer Klasse ähnlich. Monitore besitzen alle Eigenschaften eines Programmmoduls (wie in C):

- Kapselung (capsulation): Die für arbeitende Prozesse relevanten Daten müssen innerhalb des Monitors organisiert vorliegen. Der Zugriff auf Daten im Programmtext durch Monitorprozeduren macht kritische Abschnitte explizit sichtbar.

- Datenabstraktion (information hiding): Die Implementierungsdetails und das genaue strukturelle Wissen über Daten ist verborgen, sodass Prozesse sie nur über die definierten Monitorschnittstellen bearbeiten können. Dadurch können lokale Änderungen am Programmtext nur begrenzt viel im Programmablauf durcheinanderbringen.

- Bauplan (blueprint): Ein Monitor steht mit den instanziierten Exemplaren seines Typs in einem analogen Verhältnis zu einer OOP Klasse und ihren Objekten.

4.1.3 – Monitor als Betriebsmittel

Ein Monitor lässt sich als wiederverwendbares, unteilbares Betriebsmittel verstehen. In diesem Sinne ist der Monitor durch seine Funktion als Synchronisierungsmechanismus von Prozessen auch mit verschiedenen Sorten von Warteschlangen verbunden:

- Prozessinkarnationen befinden sich in der **Monitorwarteschlange**, wenn zum Zeitpunkt des Eintrittsversuchs (Prozeduraufruf) der Monitor bereits von einem anderen Prozess belegt wird. Bei Freigabe des Monitors wird einer der Prozesse in der Schlange zur Auswahl bereitgestellt.
- Prozessinkarnationen befinden sich in der **Ereigniswarteschlange**, wenn sie auf den Eintritt von Ereignissen warten, die ihre Wartebedingung aufheben. Die damit assoziierten Bedingungsvariablen sind **konsumierbare Betriebsmittel**.

Die Warteschlangen sind so implementiert, dass ein Prozess den Monitor während seiner Wartezeit nicht beeinflusst. D.h. während ein Prozess wartet, können andere Prozesse (in dem sie andere Prozeduren aufrufen) den Monitor betreten um gegebenenfalls die Wartebedingung des ersten Prozesses aufzuheben. Dadurch kann der wartende Prozess in den Monitor eintreten.

4.2 – Architektur von Monitoren

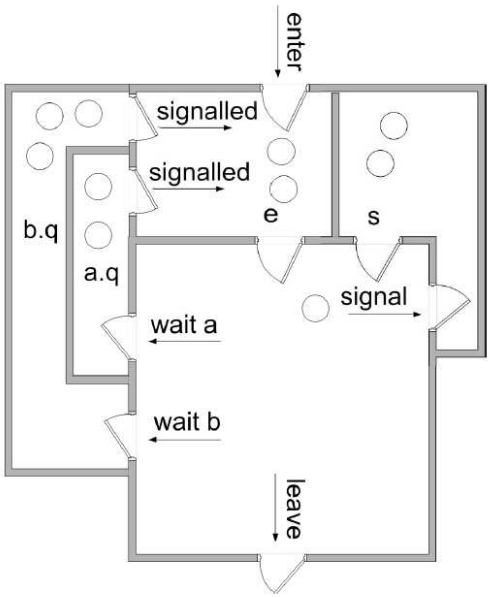
Bei der Implementierung eines Monitors kann man zwischen mehreren Modellen unterscheiden.

4.2.1 – Monitore mit blockierenden Bedingungsvariablen

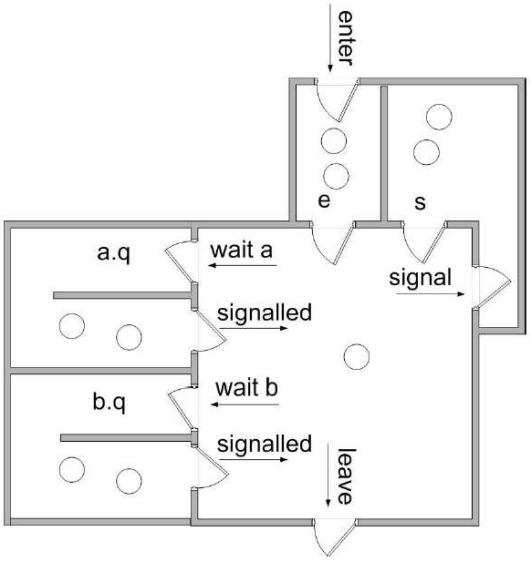
Diese Kategorie von Monitoren hat als als besonderes Merkmal, dass alle Prozesse, die die Auflösung einer Wartebedingung signalisieren, nach ihrem Signal auch in eine Monitorwarteschlange aufgenommen werden. Dabei hat der signalnehmende Prozess in der Monitorwarteschlange in der Regel einen Vorrang beim Wiedereintritt in den Monitor.

Monitor nach **Hansen**

Sobald ein Ereignis stattfindet, das eine bestimmte Wartebedingung aufhebt, werden alle Prozesse aus der entsprechenden Ereigniswarteschlange automatisch in die Monitorwarteschlange übertragen.



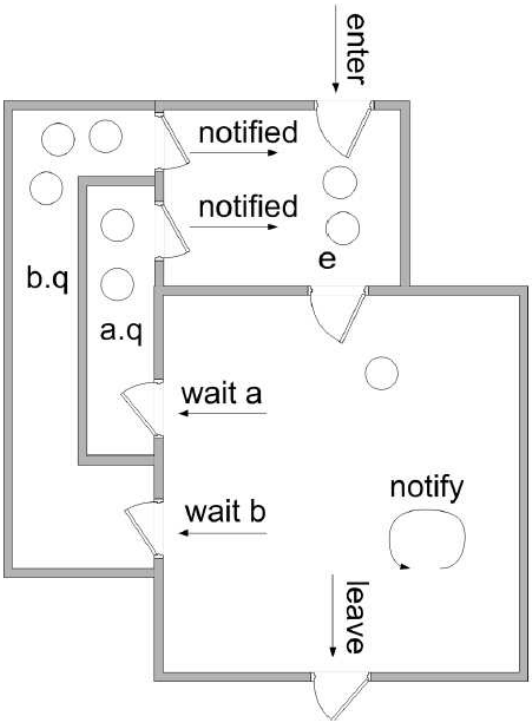
Monitor nach **Hoare**
Sobald ein Ereignis stattfindet, das eine bestimmte Wartebedingung aufhebt, darf ein Prozess aus der Ereigniswarteschlange in den Monitor eintreten.



4.2.2 – Monitore mit nichtblockierenden Bedingungsvariablen

Monitor nach **Mesa**

Sobald ein Ereignis stattfindet, das eine bestimmte Wartebedingung aufhebt, werden ein oder mehrere Prozesse aus der entsprechenden Ereigniswarteschlange automatisch in die Monitorwarteschlange übertragen. In dieser Gattung von Monitoren müssen Prozesse, deren Wartebedingung aufgelöst wurde, gegebenenfalls trotzdem auf die Freigabe des Monitors durch den signalgebenden Prozess warten, da der signalgebende Prozess nicht in die Monitorwarteschlange aufgenommen wird.



5 – Prozesssynchronisation: Semaphore u. Sperren

5.1 – Definition

Ein **Semaphor** ist eine Datenstruktur, die über folgende Eigenschaften verfügt:

- Ein **ganzzahliges Attribut s** (Integer-Variable).

- Eine Operation **P()** (abkürzung von niederl. *Prolaag*), auch bekannt als *down, wait* oder *acquire*.
  Der Aufruf von P durch einen Prozess verringert den Wert von s um 1, solange $s > 0$ gilt.
  Wenn $s \leq 0$, so blockiert der Prozess, der P() aufgerufen hat. Er kommt auf eine mit dem Semaphor assoziierte Warteliste.

- Eine Operation **V()** (abkürzung von niederl. *Verhoog*), auch bekannt als *up, signal* oder *release*.
  Der Aufruf von V durch einen Prozess erhöht den Wert von s um 1. Falls s nach diesem Aufruf einen echt-positiven Wert erhält, kann einer der blockierten Prozesse in der Warteliste des Semaphors wieder bereitgestellt werden.

P und V müssen beide auf logischer und physischer Ebene **atomar** implementiert sein. Dafür gibt es drei wesentliche Gründe. Nehmen wir an, P und V seien nicht atomar. Dann…

- …könnte die gleichzeitige Ausführung von P versehentlich mehr Prozesse passieren lassen als dies von s zugelassen wird.
- …könnte die gleichzeitige Ausführung von P und/oder V bewirken, dass s nicht dem Wert entspricht, der mit der gesamten Ausführungsanzahl beider Operationen assoziiert wird.
- …könnte das gleichzeitige Auswerten der Wartebedingung (P) und Hochzählen (V) das Schlafenlegen von Prozessen bewirken, obwohl die Wartebedingung für sie schon nicht mehr gilt („lost wake up“).

5.2 – Anwendungsszenario mit synchronisiertem Buffer

```
0  /* Prototyp eines ueblichen Semaphors*/
1  SEM *semCreate(int initVal);
2  void P(SEM *sem);
3  void V(SEM *sem);
4
5  int BUFFERSIZE;
6  char buffer[BUFFERSIZE];
7
8  unsigned_int cur = 0; /* Speichert den aktuellen Bufferindex*/
9
10
11 /* Wird bei fetch() verringert und bei store() erhöht. Wenn fetch() BUFFERSIZE
12 mal hintereinander aufgerufen wird, blockiert dieser Semaphor die weitere Aus-
13 fuerhung dieser Methode */
14 SEM full = & semCreate(n);
15
16 /* Wird bei store() verringert und bei fetch() erhöht. Wenn store() BUFFERSIZE
17 mal hintereinander aufgerufen wird, blockiert dieser Semaphor die weitere Aus-
18 fuerhung dieser Methode */
19 SEM empty = & semCreate(0);
20
21 /* Dieser binaere Semaphor definiert einen kritischen Abschnitt fuer fetch() und
22 store(), sodass keine zwei Threads je gleichzeitig eine dieser Methoden aufrufen
23 können */
24 SEM lock = & semCreate(1);
25
26 char fetch(){
27    P(&full);
28    P(&lock);
29
30    char result = buffer[cur = ++cur%BUFFERSIZE];
31
32    V(&lock);
33    V(&empty);
34
35    return result;
36 }
37
```

```
38 void store(char item){
39    P(&full);
40    P(&lock);
41
42    buffer[cur = ++cur % BUFFERSIZE] = item;
43
44    V(&lock);
45    V(&empty);
46 }
```

5.3 – Implementierung